Traduction de HOL en Dedukti

Ali Assaf

Traduction de HOL en Dedukti

Stage de M2

Master Parisien de Recherche en Informatique (MPRI)

Ali Assaf

Dates : 19 avril 2012 - 12 août 2012

Lieu : INRIA Paris-Rocquencourt / Énsiie

Encadrants :

- Guillaume Burel, Énsiie
- Gilles Dowek, INRIA Paris-Rocquencourt

Le contexte général : Les systèmes de preuve actuels (Coq, HOL, PVS, etc.) sont très utiles pour le développement des mathématiques et la vérification de programmes. Cependant, ils souffrent d'un manque d'interopérabilité qui rend difficile la réutilisation des preuves.

Le problème étudié : Dedukti est un système de preuve universel basé sur le $\lambda\Pi$-calcul modulo qui peut exprimer des preuves venant de systèmes différents. Burel et Boespflug ont déjà travaillé sur une traduction de Coq en Dedukti. L'objectif de ce stage était de concevoir une traduction de HOL vers Dedukti.

La contribution proposée : Le stage a abouti à une expression de HOL dans le $\lambda\Pi$-calcul modulo, ainsi qu'à un programme de traduction automatique des preuves de HOL, écrites dans le standard OpenTheory, vers Dedukti.

Les arguments en faveur de sa validité : Une partie simplifiée de l'encodage est prouvée comme étant correcte et complète. Le programme de traduction génère des fichiers qui sont vérifiés et validés par Dedukti.

Le bilan et les perspectives : L'ensemble de HOL et de sa bibliothèque standard est traduit et vérifié dans Dedukti. C'est une nouvelle étape pour une meilleure interopérabilité entre les systèmes de preuve.

Table des matières

1 Introduction

Le Problème

Les systèmes de preuve (Coq, HOL, PVS, etc.) sont devenus indispensables dans plusieurs domaines. En mathématiques, ils aident à prouver et à vérifier les preuves de problèmes complexes comme le théorème des quatre couleurs [13] ou la conjecture de Kepler [14]. En ingénierie du logiciel, ils permettent de vérifier la correction des programmes par rapport à leur spécification. Ceci permet d'éliminer de nombreux bugs dans des logiciels critiques comme ceux utilisés dans les centrales nucléaires, les transports, les banques, ou encore la chirurgie médicale.

Cependant, ces outils souffrent aujourd'hui d'un manque d'interopérabilité. Il est difficile de prouver un théorème dans un système et de le réutiliser dans un autre. On est souvent amené à refaire entièrement la preuve.

Pourtant, ceci n'est pas un problème pour les langages de programmation modernes. On sait aujourd'hui comment compiler un programme de façon modulaire et interfacer par exemple un bout de programme écrit en C avec un autre écrit en OCaml. Dans la perspective de l'isomorphisme de Curry-DeBruijn-Howard [18], puisque les preuves peuvent être considérés comme des programmes, il devrait être possible de faire de même pour les systèmes de preuve.

L'État de l'art

HOL représente une famille de prouveurs interactifs basés sur la théorie des types simples. Les implémentations actuelles sont HOL4, HOL Light, ProofPower, et HOL Zero. Une partie conséquente des mathématiques est formalisée dans ces systèmes, d'où l'intérêt de pouvoir réutiliser leurs preuves.

Plusieurs traductions ont été proposées pour exporter HOL vers Coq. Chantal Keller a proposé une traduction [22] dite *shallow*, dans laquelle une variable est représentée par une variable, une application par une application, etc. Cependant, elle passe par une traduction intermédiare *deep* avec des indices de DeBruijn. Au final, on obtient des théorèmes tels qu'ils seraient exprimés directement dans Coq. Plusieurs autres traductions "un-à-un" semblables ont été proposées [23], mais en général on aimerait éviter de faire n^2 traductions pour n systèmes différents.

Suivant une approche différente, Joe Hurd a développé OpenTheory [19, 20, 21], un format standard qui permet de communiquer les preuves entres différents prouveurs de la famille HOL. Le format contient les différentes instructions pour reconstruire la preuve. Malheureusement, ce format est spécifique à HOL, ce qui rend difficile l'intégration de langages extérieurs. Par exemple, Isabelle/HOL, qui est censé être assez proche du formalisme HOL, ne peut qu'importer et non exporter vers ce format.

Une troisième solution est d'exprimer les preuves dans un formalisme commun, le $\lambda\Pi$-calcul modulo [8, 6]. Ce formalisme étend le $\lambda\Pi$-calcul avec des règles de réécriture. Le système est simple mais assez expressif pour pouvoir exprimer les preuves venant de systèmes différents. Dedukti est une implémentation du $\lambda\Pi$-modulo conçue par Boespflug [5, 3, 6]. En traduisant les preuves vers ce langage, on obtient des théories exprimées dans un système commun. Alexis Dorra [9] a montré comment encoder la logique intuitionniste dans le $\lambda\Pi$-modulo. Guillaume Burel et Mathieu Boespflug ont travaillé sur une traduction de Coq en Dedukti [4].

Objectifs du stage

Ce stage se concentre sur la traduction des théorèmes de HOL en Dedukti. Cette problématique pose à la fois des problèmes théoriques et pratiques.

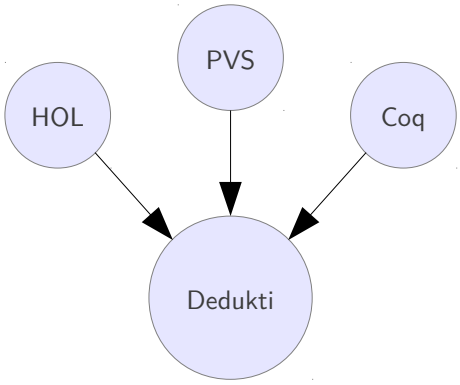

FIGURE 1 – Dedukti, un système de preuve universel

1. Quelle est la logique de HOL ? Comment l'exprimer dans le λΠ-modulo ? Peux-t-on la formaliser dans Dedukti ?
2. Comment sont représentées les preuves dans HOL ? Peux-t-on les traduire vers Dedukti ?
3. Cette traduction est-t-elle correcte ? Est-elle est complète ?

Contributions

- Encodage de la théorie des types simples dans le λΠ-calcul modulo : Pour cela je me suis basé sur les travaux de Keller et de Dorra. J'ai revu et corrigé les preuve de correction et de complétude, et je les ai adaptées à la théorie des types simples.
- Implémentation du traducteur Holide ($\sim$ 1000 lignes de code OCaml + 200 lignes de Dedukti) : Ce programme lit une preuve HOL exprimée dans le format standard OpenTheory et génère un fichier Dedukti valide.
- Amélioration de l'outil Dedukti : Au début de mon stage, Dedukti souffrait d'un manque d'utilisabilité et d'efficacité, surtout pour la taille des fichiers générés par la traduction. Tout au long du stage, j'ai collaboré étroitement avec l'équipe de développement pour améliorer son implémentation.

Plan du rapport

Dans la section 2, on va présenter Dedukti et le λΠ-calcul modulo. Dans la section 3, on va montrer comment encoder une version simplifiée du noyau logique HOL dans le λΠ-modulo et prouver que la traduction est correcte et complète. Cet encodage sera étendu dans la section 4 pour l'ensemble de HOL. Dans la section 5, on va présenter Holide, un traducteur automatique des preuves de HOL en Dedukti, ainsi que les résultats obtenus sur la traduction de la bibliothèque standard.

2 Dedukti et le λΠ-calcul modulo

Dedukti est un vérificateur de preuves universel fondé sur le formalisme du λΠ-calcul modulo [6, 8], une extension du λΠ-calcul avec des règles de réécriture. On en rappelle ici les règles générales.

2.1 Le $\lambda\Pi$-calcul

Le $\lambda\Pi$-calcul [2, 15][2, 15], aussi connu sous les noms λP et LF, est une extension du λ-calcul simplement typé avec des types dépendants. Les termes sont ceux du λ-calcul mélangés avec les types. Il n'y a pas de distinction entre les deux. Le type flèche $A \to B$ est remplacé par le *produit dépendant* $\Pi x : A.\, B$, ce qui permet d'avoir des types qui dépendent de termes comme List n, le type des listes de longueur n. On écrit $A \to B$ à la place de $\Pi x : A.\, B$ quand x n'apparaît pas dans B. Les types de types sont appelés *sortes*. Il n'en existe que deux : Type et Kind.

Définition 1 (Syntaxe du $\lambda\Pi$-calcul)**.**

$$\begin{array}{rrcl} \text{(sortes)} & s & ::= & \mathsf{Type} \mid \mathsf{Kind} \\ \text{(termes)} & t & ::= & x \mid \lambda x : t.\, t \mid t\ t \mid s \mid \Pi x : t.\, t \end{array}$$

La relation de typage est définie modulo β-équivalence grâce à une règle de conversion. Les règles de typage sont accompagnées de règles de bonne formation des contextes.

Définition 2 (Règles du $\lambda\Pi$-calcul)**.**

$$\frac{}{\emptyset \text{ well-formed}}\ \textsc{empty} \qquad \frac{\Gamma \text{ well-formed} \qquad \Gamma \vdash A : s}{\Gamma, x : A \text{ well-formed}}\ \textsc{declaration}$$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}}\ \textsc{type} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\, B : s}\ \textsc{product}$$

$$\frac{\Gamma \text{ well-formed} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{variable} \qquad \frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : [u/x]B}\ \textsc{application}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}\ \textsc{abstraction}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash t : B}\ \textsc{conversion}$$

Ce formalisme correspond, selon l'isomorphisme de Curry-DeBruijn-Howard, à la logique des prédicats, mais il est assez expressif pour encoder de nombreuses logiques différentes [15] [12]. Il permet entre autres d'encoder le système F, de façon à ce que le typage des termes soit reflété dans le $\lambda\Pi$-calcul. Cependant, la relation β n'est pas reflétée dans cet encodage. La β-réduction du système F ne correspond pas à celle du $\lambda\Pi$-calcul, d'où l'intérêt d'ajouter des règles de réécriture au $\lambda\Pi$-calcul.

2.2 Le $\lambda\Pi$-calcul modulo

Le $\lambda\Pi$-calcul peut être étendu par des règles de réécriture de la manière suivante. Soit Σ un contexte bien formé. Une règle de réécriture $\Gamma \vdash l \rightsquigarrow r$ est bien typée dans Σ quand il existe un type A tel que $\Sigma, \Gamma \vdash l : A$ and $\Sigma, \Gamma \vdash r : A$, et que toutes les variables libres de r apparaissent dans l. Un ensemble R de règles de réécriture bien typées dans Σ induit une

relation d'équivalence $\equiv_R$ sur les termes, définie comme la plus petite congruence tel que si t se réécrit en u, alors $t \equiv_R u$.

Le $\lambda\Pi$-calcul modulo (Σ, R), abrégé $\lambda\Pi$-calcul modulo R, est obtenu à partir du $\lambda\Pi$-calcul en remplaçant dans la règle de conversion la relation $\equiv_\beta$ par la relation $\equiv_{\beta R}$, c'est-à-dire la plus petite congruence qui contient les relations $\equiv_\beta$ et $\equiv_R$. Il faut aussi modifier la règle des variables pour inclure le contexte Σ, qui est appelé la *signature* du langage.

Définition 3 (Règles du $\lambda\Pi$-calcul modulo R)**.** Les règles sont celles du $\lambda\Pi$-calcul où les règles VARIABLE et CONVERSION sont remplacées par :

$$\frac{\Gamma \text{ well-formed} \qquad x : A \in \Sigma, \Gamma}{\Gamma \vdash x : A} \text{ VARIABLE}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta R} B}{\Gamma \vdash t : B} \text{ CONVERSION}$$

Exemple 1. Dans le contexte $\Sigma = [P : \mathsf{Type}, Q : \mathsf{Type}]$, le système de réécriture $\emptyset \vdash P \rightsquigarrow Q \to Q$ est bien typé. Dans le $\lambda\Pi$-calcul modulo (Σ, R), le terme $\lambda f : P.\, \lambda x : Q.\, f\ x$ est bien typé et a le type $P \to Q \to Q$.

Ce processus d'extension peut être répété plusieurs fois. Dans le $\lambda\Pi$-calcul modulo (Σ_1, R_1), si Σ_2 est un contexte bien formé et R_2 un ensemble de règles bien typées dans Σ_2, on peut définir le $\lambda\Pi$-calcul modulo (Σ_1, R_1) modulo (Σ_2, R_2), abrégé $\lambda\Pi$-calcul modulo $R_1 R_2$, et ainsi de suite.

Le $\lambda\Pi$-calcul modulo correspond, selon l'isomorphisme de Curry-DeBruijn-Howard, à la déduction modulo [11, 10], mais il peut aussi encoder de nombreux autres systèmes. En particulier, on note le résultat suivant sur les *systèmes de types purs* (PTS), qui sont une généralisation des extensions du λ-calcul simplement typé comme le système F ou le Calcul des Constructions [2].

Théorème 1. (Cousineau et Dowek [8]) *Tout PTS fonctionnel peut être encodé dans le $\lambda\Pi$-calcul modulo. L'encodage est correct et complet.*

2.3 L'outil Dedukti

Dedukti est un vérificateur de types pour le $\lambda\Pi$-calcul modulo conçu par Mathieu Boespflug [5, 3, 6]. Le langage permet de déclarer des constantes, ainsi que définir des règles de réécriture sur ces constantes.

Exemple 2. L'exemple 1 peut-être écrit dans Dedukti de la manière suivante.

```
P : Type.
Q : Type.

[] P ⇝ Q → Q.
```

Pour vérifier qu'un terme t a bien le type A, on déclare une constante qui se réécrit vers t.

```
x : P → Q → Q.
[] x ⇝ f : P ⇒ x : Q ⇒ f x.
```

Remarque. Pour que la vérification de types reste décidable, on s'impose que les systèmes de réécriture soient *confluents* et *fortement normalisants*. Par la suite, tous les systèmes examinés auront ces deux propriétés.

3 La théorie des types simples

Le noyau logique de HOL s'appelle la *théorie des types simples* (simple type theory), aussi connue sous le nom de *logique d'ordre supérieur* (higher order logic) de Church [7, 1]. Ces noms viennent du fait que les objets du langage sont les termes du lambda calcul. Il existe plusieurs versions différentes de la théorie des types simples :

- Certaines sont classiques alors que d'autres sont intuitionnistes.
- Certaines définissent l'égalité ($=$) en fonction des connecteurs logiques habituels comme l'implication ($\Rightarrow$) et la quantification universelle ($\forall$). D'autres prennent au contraire l'égalité comme connecteur de base et définissent tous les autres connecteurs logiques en fonction de l'égalité.
- Les dérivations "top-down" v.s. "bottom-up"
- Les axiomes admis peuvent être différents : extensionnalité, axiome du choix, etc.

Dans cette section, on va présenter une version simplifiée, la théorie des types simples minimale, et montrer comment l'encoder dans le $\lambda\Pi$-calcul modulo. Cela permettra d'illustrer la traduction de HOL.

3.1 La théorie des types simples minimale

Les sortes sont les types simples du lambda calcul, avec deux types de base, o et ι.

Définition 4 (Types de la théorie des types simples minimale)**.**

$$\text{(types simples)} \quad A \quad ::= \quad o \mid \iota \mid A \to A$$

Les objets sont les termes du λ-calcul simplement typé, auxquels on rajoute les constantes $\Rightarrow$ de type $o \to o \to o$ et $\forall_A$ de type $(A \to o) \to o$ pour tout type A. On distingue également au moins un élément e de type ι.

Définition 5 (Termes de la théorie des types simples minimale)**.**

$$\text{(termes simples)} \quad t \quad ::= \quad x^A \mid \lambda x^A.\, t \mid t\ t \mid \Rightarrow \mid \forall_A \mid \mathsf{e}$$

Il faut noter que les variables sont toujours annotées par leur type. Par conséquent, on peut typer les termes sans contexte. Ceci peut être problématique pour la traduction, parce qu'il n'y a pas de restriction sur les variables libres. Par la suite, on supposera que les variables ont toutes des noms différents et on omettra les annotations de type quand on n'en aura pas besoin.

Les termes de type o sont appelés *formules* et représentent les propositions de la théorie des types simples. Les termes de type $A \to o$ sont des *prédicats*. Les jugements de la logique sont donc de la forme $\Gamma \vdash \phi$ où ϕ est une formule et Γ un ensemble de formules. Les règles de dérivation sont les suivantes.

Définition 6 (Règles de la théorie des types simples minimale)**.**

$$\frac{}{\Gamma, \phi \vdash \phi}\ \text{HYP} \qquad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \Rightarrow \phi\ \psi}\ \Rightarrow\text{-INTRO} \qquad \frac{\Gamma \vdash \Rightarrow \phi\ \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}\ \Rightarrow\text{-ELIM}$$

$$\frac{\Gamma \vdash P\ x}{\Gamma \vdash \forall_A\ P}\ \forall_A\text{-INTRO}\ (x \notin \mathrm{FV}(\Gamma)) \qquad \frac{\Gamma \vdash \forall_A\ P}{\Gamma \vdash P\ t}\ \forall_A\text{-ELIM}$$

On voit bien que ces règles imitent les règles d'introduction et d'élimination de l'implication et de la quantification universelle. Ce formalisme est plus puissant que la logique des prédicats. En effet, comme on peux quantifier sur n'importe quel type A, on peux quantifier en particulier sur les propositions ou les prédicats.

Exemple 3. La proposition de la logique de second ordre $\forall ab, (a \Rightarrow (a \Rightarrow b)) \Rightarrow a \Rightarrow b$ est représentée par le terme

$$\forall_o\ (\lambda a.\ \forall_o\ (\lambda b.\ \Rightarrow\ (\Rightarrow\ a\ (\Rightarrow\ a\ b))\ (\Rightarrow\ a\ b))$$

En voici une dérivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Rightarrow a(\Rightarrow ab), a \vdash \Rightarrow a(\Rightarrow ab)}\ \text{HYP} \quad \dfrac{}{\Rightarrow a(\Rightarrow ab), a \vdash a}\ \text{HYP}}{\Rightarrow a(\Rightarrow ab), a \vdash \Rightarrow ab}\ \Rightarrow\text{-ELIM} \quad \dfrac{}{\Rightarrow a(\Rightarrow ab), a \vdash a}\ \text{HYP}}{\Rightarrow a(\Rightarrow ab), a \vdash b}\ \Rightarrow\text{-ELIM}}{\Rightarrow a(\Rightarrow ab) \vdash \Rightarrow ab}\ \Rightarrow\text{-INTRO}}{\vdash \Rightarrow(\Rightarrow a(\Rightarrow ab))(\Rightarrow ab))}\ \Rightarrow\text{-INTRO}}{\vdash \forall_o(\lambda b.\ \Rightarrow(\Rightarrow a(\Rightarrow ab))(\Rightarrow ab)))}\ \forall_o\text{-INTRO}}{\vdash \forall_o(\lambda a.\ \forall_o(\lambda b.\ \Rightarrow(\Rightarrow a(\Rightarrow ab))(\Rightarrow ab))))}\ \forall_o\text{-INTRO}$$

3.2 Encodage dans le $\lambda\Pi$-calcul modulo

Pour encoder la théorie des types simples dans le $\lambda\Pi$-calcul modulo, il faut adapter l'isomorphisme de Curry-DeBruijn-Howard [18]. Dans sa formulation usuelle, on interprète les propositions par des types et les preuves par des termes.

$$\begin{array}{ccc} \text{Propositions} & \longleftrightarrow & \text{Types} \\ \text{Preuves} & \longleftrightarrow & \text{Termes} \end{array}$$

Cependant, ceci ne marche pas dans notre cas. En effet, si on commence par déclarer deux constantes ι et o de type Type, les formules sont des termes de type o, et non pas des types. On ne peut pas non plus traduire o par Type, parce que le $\lambda\Pi$-calcul modulo ne permet pas de polymorphisme. Pour cela, on introduit une constante ε, de type $o \to \mathsf{Type}$, qui va servir à traduire les jugements de la forme $\Gamma \vdash \phi$. La correspondance devient la suivante.

$$\begin{array}{ccc} \text{Propositions} & \longleftrightarrow & \text{Termes} \\ \text{Jugements} & \longleftrightarrow & \text{Types} \\ \text{Preuves} & \longleftrightarrow & \text{Termes} \end{array}$$

Il reste encore à encoder les termes de la théorie des types simples. L'idée est d'encoder les λ-termes de la théorie des types simples par les λ-termes du $\lambda\Pi$-calcul modulo. On déclare une constante $\dot{\mathsf{e}}$ de type ι. Pour l'implication, on déclare une constante $\dot{\Rightarrow}$ de type $o \to o \to o$. Pour la quantification universelle, on rencontre un autre problème : il existe une infinité de connecteurs $\forall_A$, un pour chaque type simple A, or on ne peut pas déclarer une infinité de constantes. On ne peut pas non plus quantifier sur tous les types puisque le $\lambda\Pi$-calcul modulo ne permet pas de polymorphisme, ce qui nous empêche de déclarer une constante

$$\dot{\forall} : \Pi A : \mathsf{Type}\,.\,(A \to o) \to o$$

Pour cela, on utilise un autre encodage pour les types. On déclare un nouveau type τ, de type Type, ainsi qu'une nouvelle constante ξ de type $\tau \to \mathsf{Type}$. L'idée est d'encoder les types

simples par des termes de type τ, et d'utiliser la fonction ξ pour les interpréter comme des types. On pourra alors représenter la quantification universelle par une constante

$$\dot{\forall} : \Pi A : \tau.\, (\xi(A) \to o) \to o$$

On déclare donc deux constantes $\dot{o}$ et $\dot{\iota}$ de type τ, ainsi qu'une constante $\dot{\to}$ de type $\tau \to \tau \to \tau$. La fonction ξ est implémentée par les règles de réécriture suivantes.

$$\begin{array}{rcl} \xi(\dot{o}) & \rightsquigarrow & o \\ \xi(\dot{\iota}) & \rightsquigarrow & \iota \\ \xi(\dot{\to}\ a\ b) & \rightsquigarrow & \xi(a) \to \xi(b) \end{array}$$

D'ailleurs, on n'a même plus besoin de o et de ι puisqu'on peut les identifier avec $\xi(\dot{o})$ et $\xi(\dot{\iota})$.

Il ne reste plus qu'à implémenter la fonction ε pour interpréter l'implication et la quantification universelle par le produit dépendent.

$$\begin{array}{rcl} \varepsilon(\dot{\Rightarrow}\ p\ q) & \rightsquigarrow & \varepsilon(p) \to \varepsilon(q) \\ \varepsilon(\dot{\forall}\ A\ P) & \rightsquigarrow & \Pi x : \xi(A).\, \varepsilon(P\ x) \end{array}$$

Définition 7 ($\lambda\Pi$-calcul modulo HOL_0)**.** Soit Σ_0 le contexte contenant les symboles τ, $\dot{o}$, $\dot{\iota}$, $\dot{\to}$, ξ, $\dot{\Rightarrow}$, $\dot{\forall}$, $\dot{\mathsf{e}}$ et ε déclarés plus haut. Le système HOL_0 est obtenu à partir des règles de réécriture

$$\begin{array}{rcl} \xi(\dot{\to}\ a\ b) & \rightsquigarrow & \xi(a) \to \xi(b) \\ \varepsilon(\dot{\Rightarrow}\ p\ q) & \rightsquigarrow & \varepsilon(p) \to \varepsilon(q) \\ \varepsilon(\dot{\forall}\ A\ P) & \rightsquigarrow & \Pi x : \xi(A).\, \varepsilon(P\ x) \end{array}$$

Il est bien typé dans Σ_0.

Nous pouvons maintenant traduire la théorie des types simples minimale vers le $\lambda\Pi$-calcul modulo HOL_0. La traduction des types simples, des termes, et des contextes sont immédiates. Il est souvent utile de définir deux fonctions, $|.|$ pour la traduction vers les termes (de HOL_0) et $\|.\|$ pour la traduction vers les types (de HOL_0).

Définition 8 (Traduction des types)**.** La traduction $|A|$ des types est définie par

$$\begin{array}{rcl} |o| & = & \dot{o} \\ |\iota| & = & \dot{\iota} \\ |A \to B| & = & \dot{\to}\ |A|\ |B| \end{array}$$

On définit $\|A\| = \xi\,|A|$.

Définition 9 (Traduction des termes)**.** La traduction $|t|$ des termes est définie par

$$\begin{array}{rcl} |\Rightarrow| & = & \dot{\Rightarrow} \\ |\forall_A| & = & \dot{\forall}\ |A| \\ |x| & = & x \\ \left|\lambda x^A.\, t\right| & = & \lambda x : \|A\|\,.\ |t| \\ |t\ u| & = & |t|\ |u| \\ |\mathsf{e}| & = & \dot{\mathsf{e}} \end{array}$$

Pour toute formule ϕ, on définit $\|\phi\| = \varepsilon\,|\phi|$.

Définition 10 (Traduction des contextes)**.** Soit Γ le contexte $\phi_1, \ldots, \phi_n$. On définit

$$\|\Gamma\| = h_1 : \|\phi_1\|, \ldots, h_n : \|\phi_n\|$$

où h_1, ..., h_n sont des variables fraîches par rapport à Γ.

Pour les preuves, on applique la correspondance de Curry-DeBruijn-Howard. Une preuve de la proposition ϕ est une terme de type $\|\phi\|$. Une preuve de $\phi \Rightarrow \psi$ est une fonction qui prend en argument une preuve de ϕ et renvoie une preuve de ψ. Une preuve de $\dot{\forall}_A\ P$ est une fonction qui prend un élément x de type $\|A\|$ et qui renvoie une preuve de $P\ x$. La définition se fait par récurrence sur l'arbre de dérivation.

Définition 11 (Traduction des preuves)**.** La traduction $|\mathcal{D}|$ est définie par induction sur la structure de la dérivation.

$$\left|\overline{\Gamma, \phi \vdash \phi}^{\text{ HYP}}\right| = h \quad (\text{où } h : \|\phi\| \in \|\Gamma, \phi\|)$$

$$\left|\frac{\mathcal{D}_1}{\Gamma \vdash \Rightarrow \phi\ \psi}\ {\Rightarrow}\text{-INTRO}\right| = \lambda h : \|\phi\|\ .\ |\mathcal{D}_1| \qquad \left|\frac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Gamma \vdash \psi}\ {\Rightarrow}\text{-ELIM}\right| = |\mathcal{D}_1|\ \ |\mathcal{D}_2|$$

$$\left|\frac{\mathcal{D}_1}{\Gamma \vdash \forall_A\ P}\ \forall\text{-INTRO}\right| = \lambda x : \|A\|\ .\ |\mathcal{D}_1| \qquad \left|\frac{\mathcal{D}_1}{\Gamma \vdash P\ t}\ \forall\text{-ELIM}\right| = |\mathcal{D}_1|\ \ |t|$$

3.3 Correction de la traduction

Pour s'assurer que la traduction est correcte, il faut s'assurer que les termes produits par la traduction ont le type correct. Par exemple, la traduction d'un terme de type A doit avoir le type $\|A\|$, la traduction d'une preuve de ϕ doit avoir le type $\|\phi\|$, etc. Cependant, il faut faire attention au contexte. En effet, les termes peuvent contenir des variables libres. Par conséquent, des variables peuvent apparaître dans la preuve. Pire encore, ces variables libres peuvent même ne pas apparaître dans la conclusion de la preuve. Ce problème se pose déjà dans la logique des prédicats.

Exemple 4. En logique des prédicats, dans le contexte $\Gamma = [\forall x.P(x), \forall x.(P(x) \Rightarrow q)]$, la proposition q est prouvable,

$$\dfrac{\dfrac{\Gamma \vdash \forall x.(P(x) \Rightarrow p)}{\Gamma \vdash P(x) \Rightarrow q} \qquad \dfrac{\Gamma \vdash \forall x.P(x)}{\Gamma \vdash P(x)}}{\Gamma \vdash q}$$

mais la variable x est libre dans la preuve.

Pour cela, on introduit la notion de *contexte de termes*, qui permet de tenir compte de ces variables.

Définition 12. Un contexte de termes Δ est un contexte de la forme $x_1 : \|A_1\|, \ldots, x_n : \|A_n\|$ pour des types simples $A_1, \ldots, A_n$. Pour tout terme t (resp. contexte Γ, preuve $\mathcal{D}$), on définit Δ_t (resp. Δ_Γ, $\Delta_\mathcal{D}$) comme le contexte de termes contenant $x : \|A\|$ pour toute variable libre x de t (resp. Γ, $\mathcal{D}$).

Remarque. Un contexte de termes est toujours bien formé, et l'ordre des variables n'a pas d'importance.

Remarque. Si $\mathcal{D}$ est une preuve de $\Gamma \vdash \phi$, alors $\Delta_{\Gamma,\phi} \subseteq \Delta_\mathcal{D}$.

Théorème 2. *Pour tout type simple A, on a $\vdash |A| : \tau$ et $\vdash \|A\| : \mathsf{Type}$.*

Démonstration. On montre que $\vdash |A| : \tau$ par récurrence sur A. Puisque $\xi : \tau \to \mathsf{Type}$, on en conclut que $\vdash \|A\| : \mathsf{Type}$. □

Théorème 3. *Pour tout terme simple t de type A, on a $\Delta_t \vdash |t| : \|A\|$. Pour toute formule ϕ, on a $\Delta_\phi \vdash \|\phi\| : \mathsf{Type}$.*

Démonstration. On montre que $\Delta_t \vdash |t| : \|A\|$ par récurrence sur t. Puisque $\varepsilon : \xi(\dot{o}) \to \mathsf{Type}$, on en conclut que $\Delta_\phi \vdash \|\phi\| : \mathsf{Type}$. □

Théorème 4 (Correction)**.** *Pour toute preuve $\mathcal{D}$ de $\Gamma \vdash \phi$, on a $\Delta_\mathcal{D}, \|\Gamma\| \vdash |\mathcal{D}| : \|\phi\|$.*

Démonstration. Par récurrence sur la structure de $\mathcal{D}$. □

D'un point de vue plus pratique, on aimerait obtenir un terme clos dans le contexte vide. Pour les contextes d'hypothèses$\|\Gamma\|$, on peut quantifier sur toutes les hypothèses, de manière à transformer une preuve de $\phi_1, \ldots, \phi_n \vdash \phi$ en une preuve de $\vdash \phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow \phi$. Pour les contextes de termes Δ, il existe deux solutions.

1. On peut quantifier sur toutes les variables de Δ. Cette méthode n'est pas très élégante car elle quantifie sur des variables qui n'apparaissent pas dans la conclusion. Le théorème de l'exemple 4 deviendrait $\forall x.q$, alors que x n'apparaît pas dans q.

2. On utilise le fait que tous les types simples sont habités. On peut définir une substitution θ_Δ qui élimine toute variable libre $x : \|A\| \in \Delta$ en la remplaçant par un terme clos t de type $\|A\|$. On obtient ainsi
$$\vdash \theta(|t|) : \|A\|$$
pour les termes et
$$\theta(\|\Gamma\|) \vdash \theta(|\mathcal{D}|) : \theta(\|\phi\|)$$
pour les théorèmes.

Lemme 1. *Tous les types simples sont habités.*

Démonstration. Par récurrence sur la structure du type simple. □

Dans l'implémentation, nous avons choisi une solution intermédiaire, où on quantifie les variables qui apparaissent dans la conclusion et on élimine le reste.

3.4 Complétude de la traduction

La preuve de correction n'est pas suffisante. En effet, on pourrait traduire tout par une tautologie $\top$, et la traduction serait "correcte". On a donc besoin d'une sorte de réciproque, la *complétude* : si le type $\|\phi\|$ est habité dans le $\lambda\Pi$-calcul modulo HOL_0, alors ϕ est prouvable dans la théorie des types simples minimale.

Prouver la complétude est plus difficile que la correction. En effet, il y a des termes du $\lambda\Pi$-calcul modulo HOL_0 qui ne sont la traduction d'aucun type, terme, ou preuve de la théorie des types simples. Heureusement, on peut les réduire pour arriver à quelque chose de plus reconnaissable. Pour cette raison, la preuve de complétude se fait par inspection sur les formes normales. La normalisation forte et la préservation de type nous permettent ensuite de récupérer le résultat général.

Théorème 5. *Pour tout terme normal n, si $\vdash n : \tau$, alors $n = |A|$ pour un type simple A.*

Démonstration. Par récurrence sur la forme normale n. □

Théorème 6. *Pour tout terme normal n, si $\Delta \vdash n : \|A\|$ pour un contexte de termes Δ et un type simple A, alors $n = |t|$ pour un terme simple normal t de type A tel que $\Delta_t \subseteq \Delta$.*

Démonstration. Par récurrence sur la forme normale n. □

Théorème 7. *Pour tout terme normal n, si $\Delta, \|\Gamma\| \vdash n : \|\phi\|$ pour un contexte de termes Δ, un contexte Γ, et une formule ϕ, alors $n = |\mathcal{D}|$ pour une preuve $\mathcal{D}$ de $\Gamma \vdash \phi$ tel que $\Delta_{\mathcal{D}} \subseteq \mathcal{D}$.*

Démonstration. Par récurrence sur la forme normale n. □

4 Le système HOL

Dans cette section, on présente HOL tel qu'implémenté dans les systèmes actuels. Le système de référence pour ce stage est HOL Light [16], mais tous ces systèmes se ressemblent plus ou moins. HOL Light est implémenté par John Harrison en OCaml. Il est basé sur le formalisme de la théorie des types simples, mais diffère de la théorie qu'on a présenté dans la section précédente sur plusieurs points :

- Les types simples sont enrichis avec du polymorphisme "à la ML"
- La logique est classique
- Les connecteurs de base sont l'égalité et le symbole de choix
- Le système permet de d'introduire des nouveaux types et constantes
- Les dérivations sont "top-down"

On va donc commencer par présenter le noyau logique propre à HOL.

4.1 Le noyau logique de HOL

Les types de HOL sont les types simples, avec un polymorphisme "à la ML". On a des *variables de type*, et des *constructeur de types*.

Définition 13 (Types de HOL)**.**

$$A \quad ::= \quad \alpha \mid \mathsf{op}(A, \ldots, A)$$

Il existe deux types de base, bool et ind, exprimés comme des constructeurs de type d'arité 0. Même la flèche des types est exprimée comme un constructeur de type d'arité 2. Par la suite, on écrira $A \to B$ à la place de $\to (A, B)$. Les types bool et ind correspondent aux types o et ι de la section précédente. Les propositions de HOL sont donc les termes de type bool.

Les termes sont ceux du lambda calcul, auxquels on peux ajouter des *constantes*.

Définition 14 (Termes de HOL)**.**

$$t \quad ::= \quad x^A \mid \mathsf{c}^A \mid \lambda x^A.\, t \mid t\ t$$

Il existe deux constantes de base, l'égalité $(=)$ de type $\alpha \to \alpha \to \mathsf{bool}$, et le symbole de choix (select) de type $(\alpha \to \mathsf{bool}) \to \alpha$. On écrira $t = u$ à la place de $(=)\ t\ u$.

On peux voir le type des constantes comme un schéma de types. En effet, le terme

$$(\lambda x^{\mathsf{bool}}.\ x^{\mathsf{bool}}) = (\lambda x^{\mathsf{bool}}.\ x^{\mathsf{bool}})$$

est en fait la constante $(=)$ où la variable de type nommée α a été substituée par le type $\mathsf{bool} \to \mathsf{bool}$, appliquée deux fois au terme $\lambda x^{\mathsf{bool}}.\ x^{\mathsf{bool}}$. On notera $(=^{A \to A \to \mathsf{bool}})$ par $=_A$. Par convention, l'égalité sur les booléens s'écrit $(\Leftrightarrow)$ pour la distinguer du reste.

Il existe plusieurs versions des règles de dérivation. Celle-ci est une adaptation de la présentation de Hurd [21].

Définition 15 (Règles de HOL)**.**

$$\frac{}{\vdash t = t}\ \textsc{refl}\ t \qquad \frac{\Gamma \vdash t = u}{\Gamma \vdash \lambda x.\ t = \lambda x.\ u}\ \textsc{abs}\ x \qquad \frac{\Gamma \vdash f = g \qquad \Gamma \vdash t = u}{\Gamma \cup \Delta \vdash f\ x = g\ y}\ \textsc{app} \qquad \frac{}{\{\phi\} \vdash \phi}\ \textsc{assume}$$

$$\frac{\Gamma \vdash \phi \Leftrightarrow \psi \qquad \Delta \vdash \phi}{\Gamma \cup \Delta \vdash \psi}\ \textsc{eqMp} \qquad \frac{\Gamma \vdash \phi \qquad \Delta \vdash \psi}{(\Gamma - \{\psi\}) \cup (\Delta - \{\phi\}) \vdash \phi \Leftrightarrow \psi}\ \textsc{deductAntiSym}$$

$$\frac{}{\vdash (\lambda x.\ t)\ x = t}\ \textsc{beta}\ x\ t \qquad \frac{\Gamma \vdash \phi}{\sigma(\Gamma) \vdash \sigma(\phi)}\ \textsc{subst}\ \sigma \qquad \frac{}{\mathsf{c} = t}\ \textsc{defineConst}\ \mathsf{c}\ t$$

$$\frac{\vdash P\ t}{\vdash \mathsf{abs}\ (\mathsf{rep}\ \ y) = y \qquad \vdash P\ x \Leftrightarrow (\mathsf{rep}\ \mathsf{abs}\ x = x)}\ \textsc{defineTypeOp}\ \mathsf{op}\ \mathsf{abs}\ \mathsf{rep}$$

Les trois premières règles expriment le fait que l'égalité est une congruence réflexive. Les trois suivantes montrent comment l'égalité se comporte sur les booléens, c'est-à-dire, comme le connecteur "*si et seulement si*". Les règles BETA, et SUBST permettent de β-réduire les termes, ainsi que substituer des variables de termes et des variables de types.

Exemple 5. La transitivité de l'égalité est dérivable dans HOL : si $\Gamma \vdash x = y$ et $\Delta \vdash y = z$, alors $\Gamma \cup \Delta \vdash x = z$.

$$\dfrac{\dfrac{\dfrac{}{\vdash ((=)\ x) = ((=)\ x)}\ \textsc{refl} \qquad \Delta \vdash y = z}{\Delta \vdash (x = y) = (x = z)}\ \textsc{appThm} \qquad \Gamma \vdash x = y}{\Gamma \cup \Delta \vdash x = z}\ \textsc{eqMp}$$

Les deux dernières règles de HOL correspondent respectivement aux mécanismes de définitions de constantes et d'opérateurs de types. La règle DEFINECONST permet de définir une nouvelle constante c égale au terme clos t. Par exemple, on peut définir $\top = (\lambda x.\ x = \lambda x.\ x)$. La règle DEFINETYPEOP permet de créer, à partir d'un type de départ A et d'un prédicat P, le type des éléments de A qui satisfont P, c'est-à-dire $\{x : A \mid P\ x\}$.

Si A contient les variables de type $\alpha_1, \ldots, \alpha_n$, le nouveau type $\mathsf{op}(\alpha_1, \ldots, \alpha_n)$ sera polymorphe aussi. Ce mécanisme définit également deux fonctions, abs de type $A \to \mathsf{op}(\alpha_1, \ldots, \alpha_n)$ et rep de type $\mathsf{op}(\alpha_1, \ldots, \alpha_n) \to A$ qui permettent de passer de l'un à l'autre. Les deux conclusions de la règle expriment l'isomorphisme entre $\mathsf{op}(\alpha_1, \ldots \alpha_n)$ et $\{x : A \mid P\ x\}$, comme illustré dans la figure 2. Tous les nouveaux types de HOL (entiers naturels, types inductifs, etc.) sont construits de cette manière [17].

En plus de ces règles, HOL admet trois axiomes :

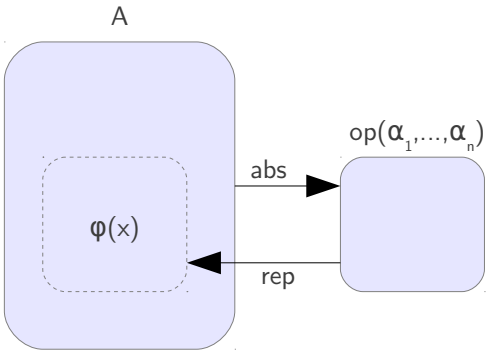

FIGURE 2 – Définition de types dans HOL

- l'η-équivalence : $\forall t.\ (\lambda x.\ t\ x = t)$
- l'axiome du choix : $\forall x.\ (p\ x \Rightarrow p\ ((\mathsf{select})\ p))$
- l'axiome de l'infini : $\exists f^{\mathsf{ind}\to\mathsf{ind}}.\ (\mathsf{injective}\ f \wedge \neg\, \mathsf{surjective}\ f)$

pour une certaine définition des connecteurs logiques $\forall$, $\Rightarrow$, $\wedge$, etc.

4.2 Encodage dans le $\lambda\Pi$-calcul modulo

Pour encoder le polymorphisme, on fait usage de l'encodage des types simples qu'on a élaboré dans la section 3.2. Une variable de type nommée α est encodée par une variable α, de type τ. Pour chaque opérateur de types op d'arité n, on déclare une constante $\dot{\mathsf{op}}$ de type

$$\underbrace{\tau \to \ldots \to \tau}_{n} \to \tau$$

qu'on écrira sous la forme plus compacte $\prod^n \tau \to \tau$. Le type $\mathsf{op}(A_1, \ldots, A_n)$ est ensuite traduit par le terme $\dot{\mathsf{op}}\ |A_1|\ \ldots\ |A_n|$.

Définition 16 (Traduction des types de HOL)**.**

$$\begin{array}{lcl} |\alpha| & = & \alpha \\ |\mathsf{op}(A_1, \ldots, A_n)| & = & \dot{\mathsf{op}}\ |A_1|\ \ldots\ |A_n| \end{array}$$

Un terme simple t de type A est toujours traduit par un terme $|t|$ de type $\|A\|$, mais $\|A\|$ peut dorénavant contenir des variables de type. La traduction est la même que la précédente, mais il faut maintenant gérer les constantes de manière générique. Pour chaque constante c de (schéma de) type A, on déclare une variable $\dot{\mathsf{c}}$ de type

$$\prod_{i=1}^{n} \alpha_i : \tau.\ \|A\|$$

où $\{\alpha_i\}_{i=1}^n$ sont les variables de types qui apparaissent dans A. Pour une instance c^B, on peut instancier $\dot{\mathsf{c}}$ en l'appliquant aux types appropriés. C'est ce qu'on avait fait pour encoder $\forall_A$ par $\dot{\forall}\ |A|$.

Cette instanciation n'est pas toujours simple à faire. Dans le cas général, comme on n'a que le type de la constante, il faut faire une opération de *filtrage* (pattern matching). Le filtrage de B par le schéma A génère, s'il réussit, une substitution $[A_i/\alpha_i]$ des variables de types (possiblement par d'autres variables de types). Si c^B est bien une instance de c^A, cette opération réussit toujours.

Définition 17 (Traduction des termes de HOL)**.**

$$\begin{array}{lcl} |x| & = & x \\ \left|\lambda x^A.\, t\right| & = & \lambda x : \|A\| \,.\, |t| \\ |t\ u| & = & |t|\ |u| \\ \left|\mathsf{c}^B\right| & = & \dot{\mathsf{c}}\ |A_1|\ \ldots\ |A_n| \quad (\text{où } \mathsf{match}(B, A) = [A_i/\alpha_i]) \end{array}$$

Il reste à traduire les preuves. Cependant, comme HOL est un système classique avec des axiomes, il n'est pas toujours possible de les traduire directement dans notre système. Pour certaines règles, il faudra déclarer des constantes, c'est-à-dire, des axiomes. Pour d'autres, on pourra les traduire directement par un terme du bon type.

L'égalité Tout dépend de la manière dont on définit l'égalité. Comme Keller [22], on a adopté l'égalité de Leibniz en ajoutant la règle de réécriture

$$\varepsilon(x =_A y) \rightsquigarrow \Pi P : \|A \to \mathsf{bool}\| \,.\, \varepsilon(P\ x) \to \varepsilon(P\ y)$$

Avec ça, on peut prouver la réflexivité REFL et la règle APP. Cependant, pour la règles ABS, on a besoin de l'*extensionnalité fonctionnelle*, qui affirme que si deux fonctions f et g de type $A \to B$ sont égales pour toutes les valeurs x de type A, alors f et g sont égales.

$$\forall fg, (\forall x, fx = gx) \Rightarrow f = g$$

On déclare pour cela une constante FUN_EXT de type

$$\Pi A : \tau.\, \Pi B : \tau.\ \|\forall fg, (\forall x, fx = gx) \Rightarrow f = g\|$$

Un des avantages de cette définition est qu'on peut également prouver l'η-extensionnalité et éviter ainsi de l'ajouter comme axiome.

Les booléens L'égalité sur les booléens se comporte comme le connecteur "si et seulement si". Avec l'égalité de Leibniz, on peut prouver la règle EQMP, mais pour DEDUCTANTISYM, on a besoin d'un axiome supplémentaire : l'*extensionnalité propositionnelle*.

$$\forall ab, (a \Rightarrow b) \Rightarrow (b \Rightarrow a) \Rightarrow a = b$$

On déclare donc une constante PROP_EXT, qu'on utilise pour prouver les règles DEDUCTANTISYM.

La β-conversion et la substitution Comme on a traduit les λ-termes de HOL par les λ-termes du $\lambda\Pi$-calcul, la βréduction sur les termes de HOL est préservée par la traduction. On a donc aucun problème à prouver la règle BETA : c'est juste la réflexivité !

Les substitutions parallèles, qu'elles soient de types ou de termes, sont elles aussi encodées par la β-réduction. Par exemple, si D est un terme qui prouve $\|\phi\|$, et σ est la substitution $[u_1/x_1, \ldots u_n/x_n]$, alors le terme

$$(\lambda x_1.\ \ldots \lambda x_n.\, D)\ u_1\ \ldots\ u_n$$

est une preuve de $\|\sigma(\phi)\|$. On note que cette méthode évite les problèmes de capture qui accompagnent généralement la notion de substitution.

Les définitions de constantes et d'opérateurs de types Comme expliqué plus haut, quand on définit une constante $\mathsf{c} = t$ dans HOL, on déclare une constante $\dot{\mathsf{c}}$ dans le $\lambda\Pi$-calcul modulo. Pour la preuve, on ajoute la règle de réécriture

$$\dot{\mathsf{c}} \rightsquigarrow |t|$$

ce qui permet de prouver $\vdash \mathsf{c} = t$ en utilisant la réflexivité. Si la constante est polymorphe, la règle devient

$$\dot{\mathsf{c}}\ \alpha_1\ \ldots\ \alpha_n \rightsquigarrow |t|$$

Les définitions d'opérateurs de types sont plus compliquées à encoder. En effet, il n'y a pas de notion native de sous-ensemble dans le $\lambda\Pi$-calcul modulo. On a pensé encoder les éléments de $\{x : A \mid \phi(x)\}$ par des paires dépendantes (x, D) d'éléments x de type $\|A\|$ accompagnés d'une preuve D de $\phi(x)$, mais cela pose plusieurs problèmes.

1. Il y a plusieurs preuves de $\phi(x)$ pour un même x. Du coup, la représentation n'est plus canonique. Ceci peut poser des problèmes de confluence.

2. Les preuves apparaissent dans les termes, qui sont normalisés par Dedukti, ce qui ralentit beaucoup le système.

Par conséquent, on a choisi une solution temporaire, avec la règle de réécriture

$$\mathsf{abs}\ (\mathsf{rep}\, y) \rightsquigarrow y$$

Ceci permet de prouver $\mathsf{abs}\ (\mathsf{rep}\, y) = y$, mais pas $\phi\ x \Leftrightarrow (\mathsf{rep}\ \ \mathsf{abs}\ \ x = x)$, qu'on pose comme axiome REP_ABS.

Les axiomes Enfin, les axiomes restants (comme l'axiome du choix ou l'axiome de l'infini) sont posés comme axiomes dans le $\lambda\Pi$-calcul modulo. Au final, on encode le noyau de HOL, qui est un système classique avec 3 axiomes, dans un système intuitionniste avec 5 axiomes.

4.3 Flexibilité de la traduction

Dans HOL, les connecteurs logiques sont définis à partir de l'égalité.

$$\begin{array}{lcl} \top & = & (\lambda x.\, x) = (\lambda x.\, x) \\ \wedge & = & \lambda p.\, \lambda q.\, (\lambda f.\, f\ p\ q) = (\lambda f.\, f\ \top\ \top) \\ \Rightarrow & = & \lambda p.\, \lambda q.\, p \wedge q \Leftrightarrow p \\ \forall & = & \lambda P.\, P = \lambda x.\, \top \\ & \ldots & \end{array}$$

On peut montrer que les règles d'introduction et d'élimination usuelles des connecteurs sont dérivables à partir de ces définitions. Elles ne correspondent pas aux définitions "naturelles" de l'implication et de la quantification universelle de la théorie des types simples qu'on a présentées dans la section 3. Si on voulait connecter ces preuves avec celles de Coq par exemple, on aurait deux définitions différentes des connecteurs logiques. De plus, les définitions naturelles sont plus légères en réécriture et donc plus efficaces [1].

Heureusement, il est possible de contourner ces définitions. Si on préfère définir une constante c par t' au lieu de t, on peut réécrire c vers t' à la place de t. Pour s'assurer que cela ne va pas

1. On a observé un gain de performance (jusqu'à 20 fois plus vite) pour la vérification sur un interpréteur de Dedukti.

causer de problèmes avec le reste des théorèmes, il suffit juste de dériver la règle DEFINECONST. En d'autres termes, il faut prouver $\mathsf{c} = t$ dans Dedukti.

Pour les connecteurs logiques, on a développé une méthode générique pour prouver cette équivalence :

1. On prouve les règles d'introduction et d'élimination des deux définitions.
2. On prouve que chacune implique l'autre, en utilisant les règles d'élimination de la première et d'introduction de la seconde.
3. On en déduit que les deux propositions sont égales par l'extensionnalité propositionnelle.
4. On conclut que les deux connecteurs sont égaux par l'extensionnalité fonctionnelle (s'ils prennent des arguments).

Exemple 6. Pour prouver que deux définitions and_1 et and_2 de la conjonction sont équivalentes, on commence par dériver les règles d'introduction et d'élimination de chacune. On prouve que $\mathsf{and}_1\ p\ q$ implique $\mathsf{and}_2\ p\ q$. Pour cela, on applique les règles d'élimination sur l'hypothèse $\mathsf{and}_1\ p\ q$ pour obtenir p et q, que l'on utilise dans la règle d'introduction pour obtenir $\mathsf{and}_2\ p\ q$. De même pour la réciproque. On en déduit que $\mathsf{and}_1\ p\ q \Leftrightarrow \mathsf{and}_2\ p\ q$ par l'extensionnalité propositionnelle [2]. On conclut que $\mathsf{and}_1 = \mathsf{and}_2$ en deux coups d'extensionnalité fonctionnelle.

On peut faire de même pour les définitions d'opérateurs de types. On pourrait par exemple vouloir traduire les entiers naturels de HOL par les entiers naturels de Coq, qui sont construits d'une manière très différente. Encore une fois, il est possible de contourner la définition de HOL. Il suffit de dériver les deux théorèmes de la règle DEFINETYPEOP.

De manière générale, on remarque que la traduction est entièrement indépendante de la manière dont on définit les connecteurs logiques, les constantes, et les types. La traduction reste correcte dans la mesure où l'on on se conforme à l'interface de l'encodage, c'est à dire qu'on traduit un type simple par un terme de type τ et un terme simple de type A par un terme de type $\|A\|$, et qu'on prouve les règles de HOL correspondantes (ou qu'on les pose en axiomes).

5 Le traducteur automatique Holide

Holide est un traducteur automatique de HOL en Dedukti écrit en OCaml. Le programme est disponible en ligne sur le site de Dedukti [5]. Il utilise l'encodage qu'on a présenté dans la section précédente. On peut distinguer trois étapes :

1. Récupération des théorèmes de HOL
2. Traduction des théorèmes en Dedukti
3. Validation des théorèmes dans Dedukti

5.1 Les preuves dans HOL

Le système HOL Light est implémenté en OCaml suivant une architecture dite *LCF*. Concrètement, il consiste d'un module principal qui est chargé au lancement, et c'est OCaml qui prends le rôle de métalangage dans lequel on prouve les théorèmes. D'ailleurs, c'est de là que vient le nom ML (métalangage).

2. On rappelle que $\Leftrightarrow$ est l'égalité sur les booléens.

Le module contient le noyau logique de HOL. Il définit un type `thm` qui représente les théorèmes. La particularité du système vient du fait que `thm` est un type abstrait : on n'a pas accès aux constructeurs de ce type, on y a juste accès a travers l'interface des fonctions prédéfinies dans le module. Ces fonctions correspondent exactement aux règles de dérivation de HOL. Par exemple, il y a une fonction `REFL` de type `term` $\rightarrow$ `thm`. Pour la sûreté du système, il suffit juste d'avoir confiance en la correction du noyau. Tous les théorèmes seront corrects par construction.

Une des conséquences majeures de ce modèle est que les preuves ne sont pas gardées en mémoire. Pour récupérer les preuves, on est donc obligés de modifier le noyau afin d'observer les règles utilisées durant la construction des théorèmes. Plusieurs modifications ont été proposées.

- ProofRecording : Steven Obua [23] a modifié le noyau pour exporter les preuves vers des fichiers XML. Ces fichiers contiennent les arbres de preuve étiquetés avec le nom des règles utilisées. C'est cette extension que Chantal Keller a adapté pour exporter les preuves vers Coq [22].
- OpenTheory : Joe Hurd [19, 20, 21]a proposé ce format standard de preuves et de théories pour les systèmes HOL. Il a modifié le noyau pour pouvoir exporter les preuves vers ce format. Une preuve de OpenTheory consiste d'un fichier texte qui contient une liste d'instructions pour reconstruire la preuve.

Pour ce stage, on a choisi de se baser sur OpenTheory. Premièrement, ProofRecording n'est plus mis à jour, alors que OpenTheory est toujours développé. Deuxièmement, ce standard permet de récupérer les preuves d'autres systèmes comme HOL4 ou ProofPower. Enfin, il définit une bibliothèque standard de théorèmes, commune à tous les systèmes HOL. La bibliothèque est organisée en modules appelés *paquets* (packages), chacun ayant un thème particulier. Par exemple, le paquet `bool` contient les définitions des connecteurs booléens ansi que tous les théorèmes associés, le sous-paquet `bool-class` inclut les théorème classiques, etc.

Cependant, OpenTheory a aussi ses inconvénients. Le format ne contient que les règles de bases. Toutes les règles dérivées sont complètement expansées en règles élémentaires. Ceci pose problème par exemple pour les définitions des connecteurs logiques. D'autre part, les théorèmes n'ont pas de noms. Par conséquent, il est difficile de faire le lien avec un théorème qu'on a prouvé précédemment.

5.2 L'outil Holide

Le programme lit un fichier de preuve (*article file*) du format OpenTheory et génère un fichier Dedukti.

```
holide bool.art -o bool.dk
```

Durant la traduction, le programme exécute les instructions pour reconstruire les termes et les preuves en mémoire. Il traduit et imprime les théorèmes au fur et à mesure. Pour cela, il faut reconstruire les types des termes, renommer les variables afin d'éviter les collisions, quantifier les variables libres, etc.

Si on reconstruit naïvement la preuve, on obtient des termes de preuve gigantesques qu'on ne pourra pas enregistrer ou vérifier. En effet, Steven Obua [23] et Chantal Keller [22] avaient déjà remarqué qu'il faut utiliser une forme de partage. OpenTheory partage les sous-preuves communes en utilisant un dictionnaire pouvant être réutilisé pour la traduction, mais on a fait mieux.

Pour chaque étape de dérivation, on crée une nouvelle constante qui représente cette étape intermédiaire. Par la suite, on pourra faire référence à cette étape en utilisant son nom. Cette

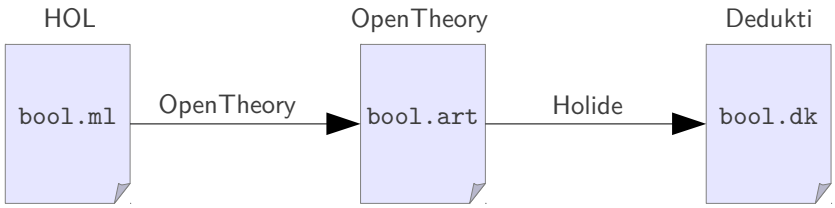

FIGURE 3 – Processus de traduction de HOL en Dedukti

transformation en étapes élémentaires nommées est similaire aux représentations intermédiaires utilisées dans les techniques de compilation comme la *static single assignment* (SSA) ou la *continuation passing style* (CPS). En pratique, on a observé que cette transformation produit un gain de performance important dans le temps de traduction et de vérification, jusqu'à 30 fois plus rapide. Certains termes de preuves étaient même tellement grands que le programme échouait sans cette optimisation. Cette transformation linéarise les arbres de preuve et permet d'ignorer complètement leur profondeur.

Exemple 7. L'exemple 5 est traduit de la manière suivante.

```
...

step_1 : true (eq (arr A bool) (eq A x) (eq Ax)).
[] step_1 ↪ REFL (arr A bool) (eq A x).

step_2 : true (eq bool (eq A x y) (eq A x z)).
[] step_2 ↪ APP_THM A bool (eq A x) (eq A x) x z step_1 hyp_2.

step_1 : true (eq A x z).
[] step_3 ↪ EQ_MP (eq A x y) (eq A x z) hyp_1.
```

Le fichier obtenu peut être vérifié par Dedukti. Il dépend d'un fichier `hol.dk`, qui contient la formalisation du noyau HOL tel qu'on l'a montré précédemment. Ce fichier définit les types, termes, et jugements de HOL, ainsi que les axiomes et règles de dérivations élémentaires.

```
dedukti hol.dk bool.dk | lua -l dedukti -
```

5.3 Résultats

L'ensemble de la librairie standard de OpenTheory a été traduite. Les fichiers générés sont très gros, leur taille pouvant atteindre plusieurs centaines de mégaoctets. Chantal Keller avait déjà fait cette observation pour sa traduction. Ceci vient de plusieurs raisons :

- Il n'y a pas d'arguments implicites dans Dedukti. Par conséquent on doit souvent ajouter des annotations de type.
- On ré-affiche les propositions à chaque étape intermédiaire.
- La β-réduction et l'expansion de constantes sont explicites dans les preuves de HOL. Les preuves ne sont pas *calculatoires*. Elles doivent montrer toutes les étapes de calcul et peuvent être très longues.

Paquet	Taille OpenTheory (en Ko)	Temps de traduction (en s)	Taille Dedukti (en Ko)	Partie vérifiée	Temps de vérification (en s)
bool	305	2	4938	100%	58
function	89	1	2300	100%	15
list	1377	39	86720	66%	441
natural	1952	38	93378	45%	698
option	520	18	39636	100%	33
pair	195	4	6825	100%	67
real	1754	46	126786	1%	5
relation	971	49	76644	21%	132
set	2329	65	119573	28%	286
sum	502	17	39384	0%	0
unit	26	0	442	100%	3
Total	10020	280	596626	35%	1,739

Tableau 1 – Résultats de la traduction des paquets OpenTheory en Dedukti

On a essayé de vérifier les fichiers générés dans Dedukti. Ceci n'a pas toujours été possible à cause de leur taille conséquente. Les fichiers Dedukti sont près de 50 fois plus gros que ceux de OpenTheory, et prennent beaucoup de temps à vérifier. Pour les plus gros, il n'y avait pas assez de mémoire vive sur la machine. Néanmoins, tous les fichiers de petite taille ont été validés avec succès. Ceci permet de vérifier encore une fois la correction de la traduction. Ces tests on été réalisés sur une machine Intel Xeon 2.67GHz à 4 Go de RAM. Les résultats sont affichés dans le tableau 1. Ils sont comparables à ceux obtenus par Chantal Keller [22] en terme de temps et de mémoire vive utilisée pour la vérification.

6 Conclusion

On a présenté une traduction complète de HOL en Dedukti. On a montré comment encoder le noyau logique dans le $\lambda\Pi$-calcul modulo et prouvé que l'encodage est correct et complet. Côté implémentation, l'outil Holide traduit l'ensemble de la bibliothèque standard de HOL et produit des fichiers Dedukti valides.

Comparé au travail de Chantal Keller, cette traduction est directe, sans passer par une traduction *deep*. D'autre part, on montre comment encoder le polymorphisme dans un système qui n'en a pas, tout en reflétant la β-réduction.

Travail futur et perspectives

On peut améliorer l'implémentation actuelle en ajoutant des liens entre les paquets. En effet, parce que les théorèmes sont anonymes dans OpenTheory, il n'est pas possible de faire référence à un théorème prouvé dans un autre paquet. Ainsi, les théorèmes prouvés dans la bibliothèque `bool` sont définis comme axiomes dans la bibliothèque `function`. Pour résoudre ce problème, OpenTheory inclut une couche supplémentaire de composition de paquets. Les fichiers `.thy` (*theory files*) contiennent les informations nécessaires pour importer les théorèmes d'un paquet comme axiomes d'un autre. Il faudra donc inclure ces mécanismes de composition dans la traduction afin d'éviter de déclarer des axiomes inutiles.

La prochaine étape sera de combiner cette traduction avec celle de Coq. Cependant, la

réunion naïve de ces deux formalismes est incohérente. En effet, HOL suppose que tous les types sont habités, alors que Coq permet de créer des types vides. Dans HOL, ceci est exprimé par la présence du symbole de choix select, de type $(\alpha \to \mathsf{bool}) \to \alpha$. Pour éviter toute contradiction, il faudra modifier le type du symbole de choix en $\alpha \to (\alpha \to \mathsf{bool}) \to \alpha$ afin d'empêcher son application à un type vide. Il faudra également modifier les énoncés des théorèmes qui contiennent des variables de types pour s'assurer qu'on ne quantifie que les types non vides.

Ce travail peut servir de base pour traduire d'autres systèmes comme PVS, dont les mécanismes de définition de types sont similaires à ceux de HOL mais qui permettent d'avoir des types dépendants. On pourra aussi essayer de concevoir des traductions inverses, de façon à pouvoir faire remonter les preuves de Dedukti vers les autres prouveurs. A long terme, on obtiendra un moyen de connecter les preuves des différents systèmes, ce qui leur permettra de mieux interagir ensemble.

Remerciements

Je remercie Guillaume Burel et Gilles Dowek pour m'avoir supervisé durant ce stage, ainsi que Mathieu Boespflug pour ses remarques et conseils.

Références

[1] Peter B. Andrews. *An introduction to mathematical logic and type theory : to truth through proof*. Academic Press, Inc., 1986.

[2] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, 1992.

[3] Mathieu Boespflug. *Conception d'un noyau de vérification de preuves pour le lambda-Pi-calcul modulo*. PhD thesis, 2011.

[4] Mathieu Boespflug and Guillaume Burel. CoqInE : Translating the calculus of inductive constructions into the λΠ-calculus modulo. In David Pichardie and Tjark Weber, editors, *PxTP*, 2012.

[5] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. Dedukti. `https://www.rocq.inria.fr/deducteam/Dedukti/index.html`.

[6] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *PxTP*, 2012.

[7] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2), 1940.

[8] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Typed Lambda Calculi and Applications*, 2007.

[9] Alexis Dorra. Equivalence de Curry-Howard entre le lambda-Pi-calcul et la logique intuitionniste. Rapport de stage de L3, LIX, Ecole Polytechnique, 2011.

[10] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, October 2003.

[11] Gilles Dowek and Benjamin Werner. Proof normalization modulo. In *International Workshop on Types for Proofs and Programs*, 1998.

[12] Herman Geuvers and Erik Barendsen. Some logical and syntactical observations concerning the first-order dependent type system λP. *Mathematical Structures in Computer Science*, 9(4), August 1999.

[13] Georges Gonthier. Formal proof – the four colour theorem. In *Asian Symposium on Computer Mathematics*, 2007.

[14] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the kepler conjecture. *Discrete & Computational Geometry*, 44, 2010.

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40, 1993.

[16] John Harrison. *The HOL Light System Reference.*

[17] John Harrison. Inductive definitions : Automation and application. In *International Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1995.

[18] William A. Howard. The formulae–as–types notion of construction. In *Essays on Combinatory Logic, Lambda Calculus, and Formalism.* 1980.

[19] Joe Hurd. OpenTheory : Package management for higher order logic theories. In *Programming Languages for Mechanized Mathematics Systems*, 2009.

[20] Joe Hurd. Composable packages for higher order logic theories. In *Verification Workshop*, 2010.

[21] Joe Hurd. The OpenTheory standard theory library. In *Nasa Formal Methods*, 2011.

[22] Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In *Interactive Theorem Proving*, 2010.

[23] Steven Obua and Sebastian Skalberg. Importing hol into isabelle/hol. In *International Joint Conference on Automated Reasoning*, 2006.